mutex(3T)

NAME

mutex - concepts relating to mutual exclusion locks

DESCRIPTION

FUNCTION	ACTION
mutex_init	Initialize a mutex.
mutex_destroy	Destroy a mutex.
mutex_lock	Lock a mutex.
mutex_trylock	Attempt to lock a mutex.
mutex_unlock	Unlock a mutex.
pthread_mutex_init	Initialize a mutex.
pthread_mutex_destroy	Destroy a mutex.
pthread_mutex_lock	Lock a mutex.
pthread_mutex_trylock	Attempt to lock a mutex.
pthread_mutex_unlock	Unlock a mutex.

Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code which access shared data (that is, mutexes are used to serialize the execution of threads). All mutexes must be global. A successful call to acquire a mutex will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks the mutex.

Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes (see **mmap(2)**), and have been initialized for this task.

Initialization

Mutexes are either intra-process or inter-process, depending upon the argument passed implicitly or explicitly to the initialization of that mutex. A statically allocated mutex does not need to be explicitly initialized; by default, a statically allocated mutex is initialized with all zeros and its scope is set to be within the calling process.

For inter-process synchronization, a mutex needs to be allocated in memory shared between these processes. Since the memory for such a mutex must be allocated dynamically, the mutex needs to be explicitly initialized with the appropriate attribute that indicates inter-process use.

Locking and Unlocking

A critical section of code is enclosed by a call to lock the mutex and the call to unlock the mutex to protect it from simultaneous access by multiple threads. Only one thread at a time may possess mutually exclusive access to the critical section of code that is enclosed by the mutex-locking call and the mutex-unlocking call, whether the mutex's scope is intra-process or inter-process. A thread calling to lock the mutex either gets exclusive access to the code starting from the successful locking until its call to unlock the mutex, or it waits until the mutex is unlocked by the thread that locked it.

Mutexes have ownership, unlike semaphores. Only the thread that locked a mutex, (that is, the owner of the mutex), should unlock it.

If a thread waiting for a mutex receives a signal, upon return from the signal handler, the thread resumes waiting for the mutex as if there was no interrupt.

Caveats

Mutexes are almost like data - they can be embedded in data structures, files, dynamic or static memory, and so forth. Hence, they are easy to introduce into a program. However, too many mutexes can degrade performance and scalability of the application. Because too few mutexes can hinder the concurrency of the application, they should be introduced with care. Also, incorrect usage (such as recursive calls, or violation of locking order, and so forth) can lead to deadlocks, or worse, data inconsistencies.

ATTRIBUTES

See **attributes(5)** for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

SEE ALSO

mmap(2), **shmop(2)**, **mutex destroy(3T)**, **mutex init(3T)**, **mutex lock(3T)**, **mutex trylock(3T)**, **mutex unlock(3T)**, **pthread mutex destroy(3T)**, **pthread mutex init(3T)**, **pthread mutex lock(3T)**, **pthread mutex trylock(3T)**, **pthread mutex unlock(3T)**, **pthread create(3T)**, **pthread mutexattr init(3T)**, **attributes(5)**, **standards(5)**

NOTES

In the current implementation of threads, pthread_mutex_lock(), pthread_mutex_unlock(), mutex_lock() mutex_unlock(), pthread_mutex_trylock(), and mutex_trylock() do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return EINVAL. Interfaces for mutexes with an invalid type have unspecified behavior.

By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined.

USYNC_THREAD does not support multiple mapplings to the same logical synch object. If you need to mmap() a synch object to different locations within the same address space, then the synch object should be initialized as a shared object USYNC_PROCESS for Solaris, and PTHREAD_PROCESS_PRIVATE for POSIX.

Man(1) output converted with man2html